A Multi-Encoding Approach for LTL Symbolic Satisfiability Checking

Kristin Y. Rozier and Moshe Y. Vardi

NASA Ames Research Center

Rice University

June 24, 2011

Model Checking

Model Checking:

1. Create a system model with formal semantics, M.
2. Encapsulate desired properties in a formal specification, f.
3. Check that M satisfies f.

Model checking finds disagreements between
the system model and the formal specification.

Model Checking

Model Checking:

1. Create a system model with formal semantics, M.
2. Encapsulate desired properties in a formal specification, f.
3. Check that M satisfies f.

Model checking finds disagreements between the system model and the formal specification.

Successful industrial adoption!

Model Checking

Model Checking:

1. Create a system model with formal semantics, M.
2. Encapsulate desired properties in a formal specification, f.
3. Check that M satisfies f.

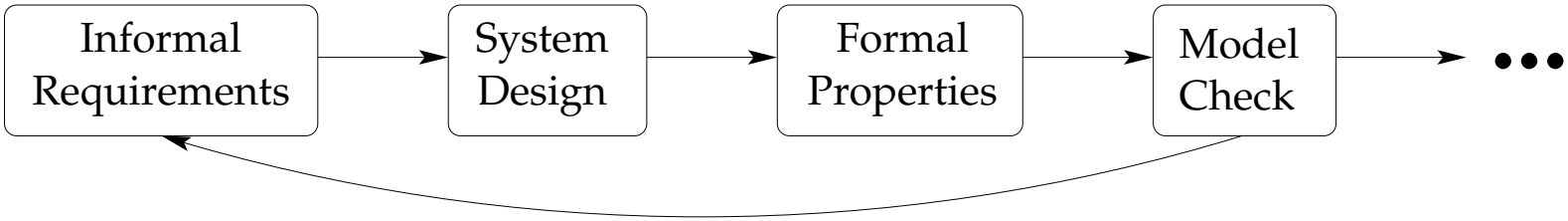
Model checking finds disagreements between the system model and the formal specification.

Successful industrial adoption!

Requires writing formal properties!

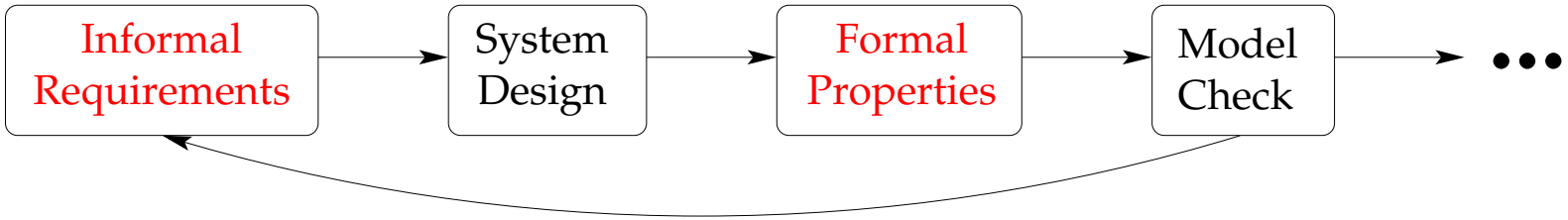
Property-Based Design

Software Development Cycle:



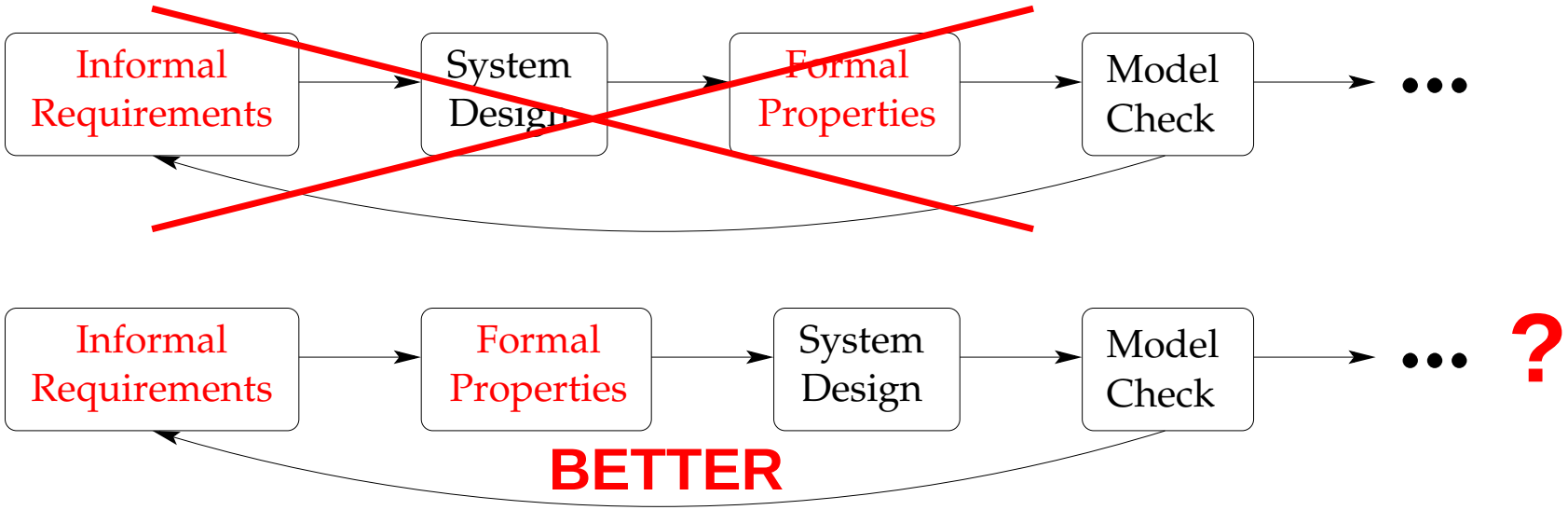
Property-Based Design

Software Development Cycle:



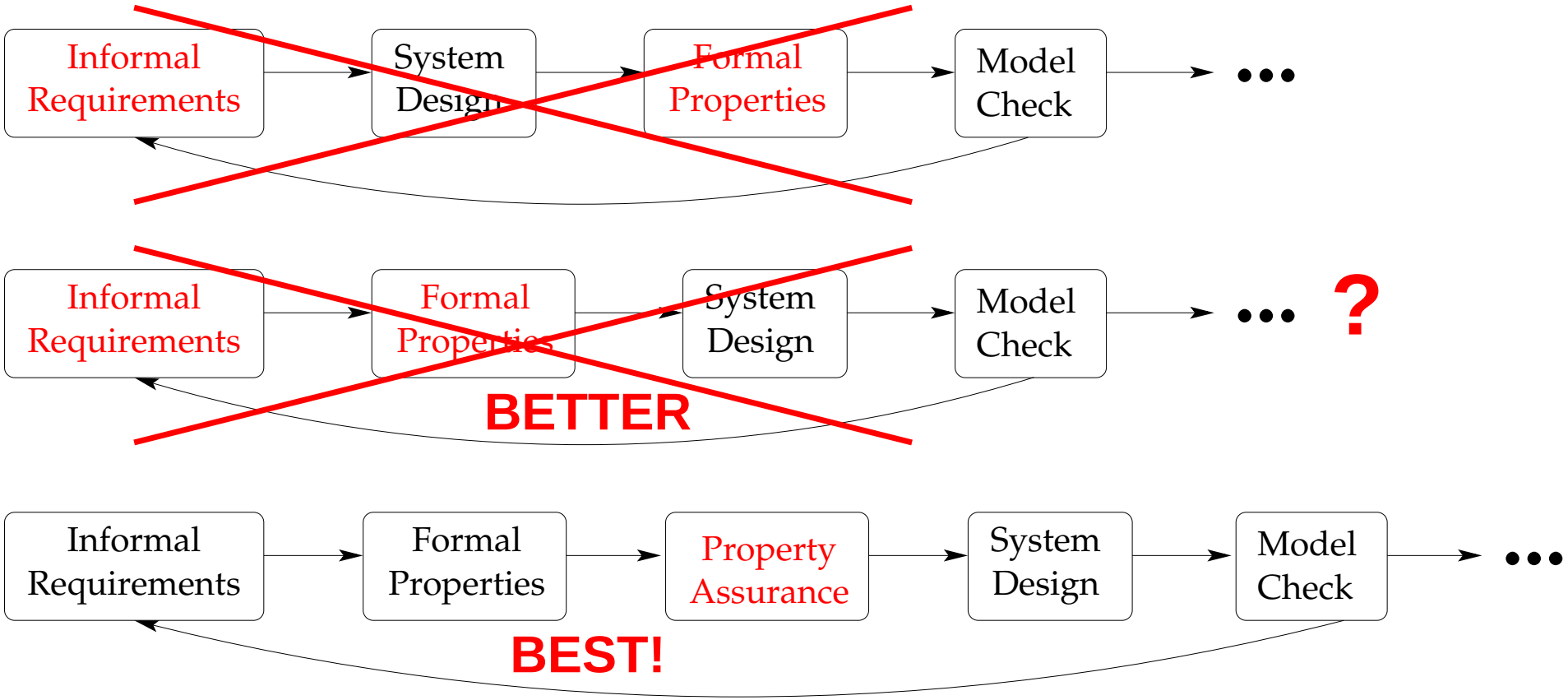
Property-Based Design

Software Development Cycle:



Property-Based Design

Software Development Cycle:



Property Assurance: Satisfiability Checking

$M \models f$ may not mean the system has the intended behavior

Recall that if a property f is *valid* then $\neg f$ is *unsatisfiable*.

If $\neg f$ is not satisfiable, then

- There can never be a counterexample.
- Model checkers will always return "success."
- f is probably wrong.

Property Assurance: Satisfiability Checking

$M \models f$ may not mean the system has the intended behavior

$M \not\models f$ may not mean the system does not have the intended behavior

Recall that if a property f is *valid* then $\neg f$ is *unsatisfiable*.

If $\neg f$ is not satisfiable, then

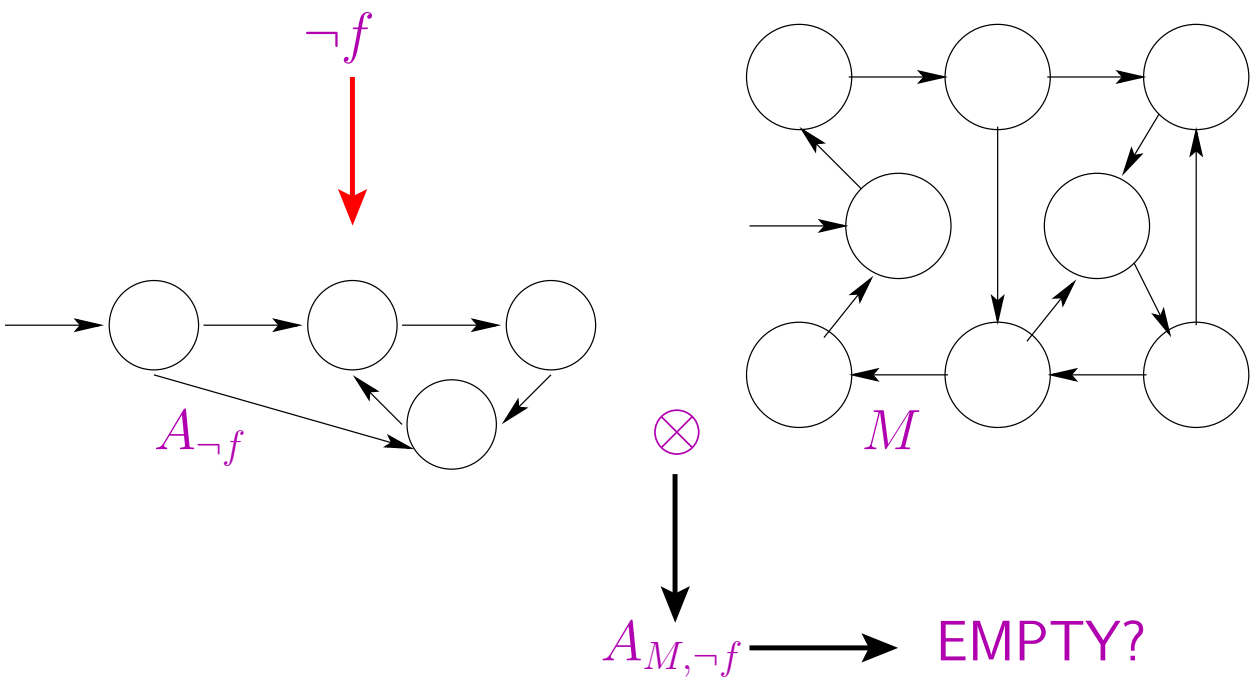
- There can never be a counterexample.
- Model checkers will always return "success."
- f is probably wrong.

If f is not satisfiable, then

- There is always a counterexample.
- Model checkers will always return "failure."
- f is probably wrong.

Automata-Theoretic Approach to Model Checking

Requires efficient LTL-to-automaton translation.



LTL Satisfiability Checking Reduces to Model Checking

- Let property f be a formula over the set *Prop* of propositions.
- Let the system model M be *universal*. That is, it contains all possible traces over *Prop*.
- Then f is satisfiable precisely when M does *not* satisfy $\neg f$.

It *should* be easy to add an LTL Satisfiability Checking feature to all model checking tools!

For each property f and $\neg f$ we should check for satisfiability.

LTL Satisfiability Using SMV

1. Model check $\neg f$ against a *universal SMV model*.

```
MODULE main
  VAR
    a : boolean;
    b : boolean;
    c : boolean;
  LTLSPEC !f
  FAIRNESS
    1
```

2. SMV:
   1. Negates the property, $\neg f$.
   2. Symbolically compiles f into A_f and conjoins A_f with the universal model.
   3. Searches for a fair path that satisfies f.

LTL-to-Automaton Complexity

- LTL property of size m
- LTL satisfiability checking takes time $2^{O(m)}$.
- LTL-to-automata translation has dramatic impact on satisfiability check.

Two approaches to satisfiability checking:

1. *explicit* automaton construction & emptiness check
2. *symbolic* automaton construction & emptiness check

LTL-to-Automaton Complexity

- LTL property of size m
- LTL satisfiability checking takes time $2^{O(m)}$.
- LTL-to-automata translation has dramatic impact on satisfiability check.

Two approaches to satisfiability checking:

1. *explicit* automaton construction & emptiness check: *highly* studied
2. *symbolic* automaton construction & emptiness check

LTL-to-Automaton Complexity

- LTL property of size m
- LTL satisfiability checking takes time $2^{O(m)}$.
- LTL-to-automata translation has dramatic impact on satisfiability check.

Two approaches to satisfiability checking:

1. *explicit* automaton construction & emptiness check: *highly* studied
2. *symbolic* automaton construction & emptiness check: *hardly* studied

LTL Satisfiability Checking via Symbolic Model Checking

$f = (p \mathcal{U} q)$

```
module main() {
  VAR
    p: boolean;
    q: boolean;
    EL_X__p_U_q : boolean;
  DEFINE S__p_U_q := q | (p & EL_X__p_U_q);
  TRANS  ( EL_X__p_U_q = (next(S__p_U_q) ))
  FAIRNESS (!S__p_U_q | q)
  SPEC !(S__p_U_q & EG TRUE) }
```

symbolic $A_{\neg f}$

```
module main() {
  VAR
    p: boolean;
    q: boolean;
  FAIRNESS  TRUE; }
```

universal M

$\otimes$

$A_{M,\neg f}$  CadenceSMV:EMPTY?

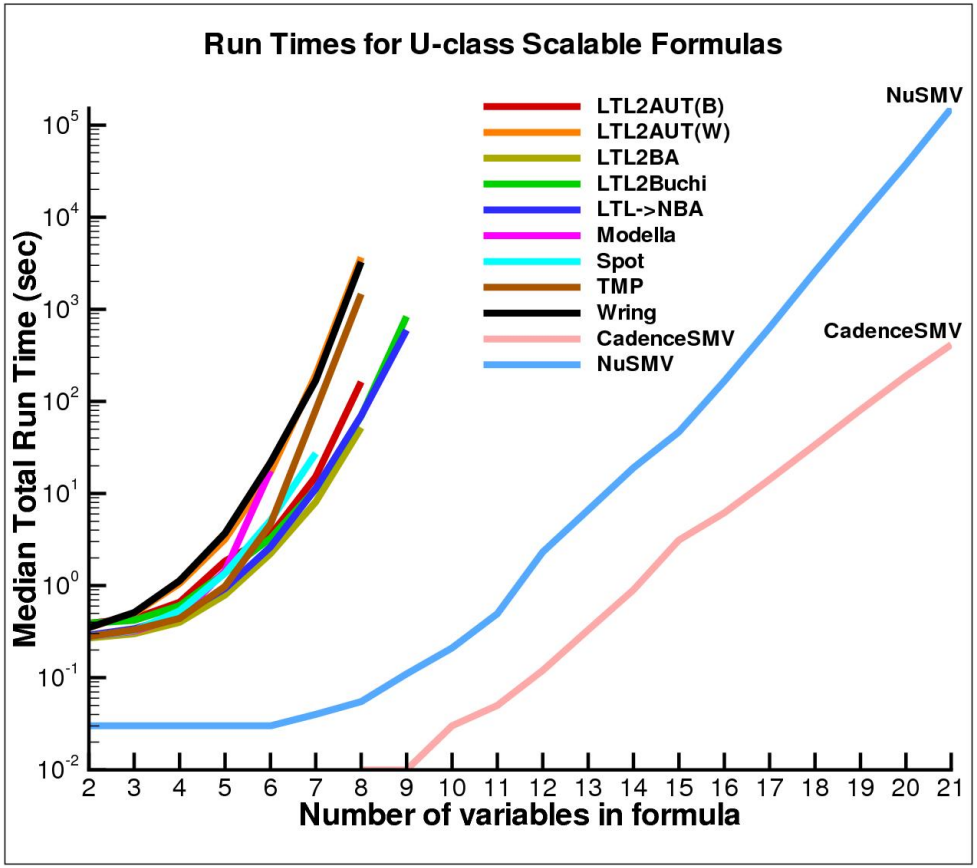
The encoding of $A_{\neg f}$ has a major impact on complexity.

Satisfiability Checking Implementation

The symbolic approach is vastly superior to the explicit approach.[1]

Symbolic Model Checkers:

- Representation: using Boolean formulas
- Analysis: using Binary Decision Diagrams (BDDs)



[1] Kristin Y. Rozier and Moshe Y. Vardi, LTL Satisfiability Checking. SPIN'07.

The ONE Symbolic Encoding

Since 1997, there is only one encoding for LTL-to-symbolic automata:
Clarke, Grumberg, and Hamaguchi (CGH).

The ONE Symbolic Encoding

Since 1997, there is only one encoding for LTL-to-symbolic automata:
Clarke, Grumberg, and Hamaguchi (CGH).

Can we do it differently?

The ONE Symbolic Encoding

Since 1997, there is only one encoding for LTL-to-symbolic automata:
Clarke, Grumberg, and Hamaguchi (CGH).

Can we do it differently?

Can we do it better?

The ONE Symbolic Encoding

Since 1997, there is only one encoding for LTL-to-symbolic automata:
Clarke, Grumberg, and Hamaguchi (CGH).

Can we do it differently?

Can we do it better?

YES!!! Exponentially better!

A Set of 30 Symbolic Automata Encodings

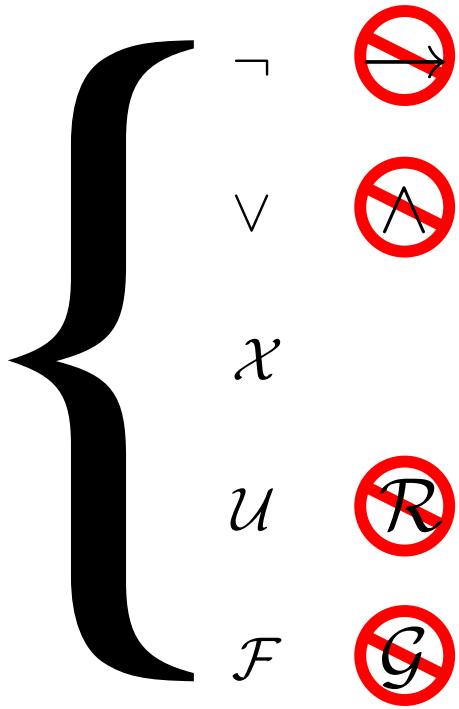
Our novel encodings are combinations of four components:

1. Normal Form: BNF or NNF
2. Automaton Form: GBA or TGBA
3. Transition Form: fussy or sloppy
4. Variable Order: default, naïve, `LEXP`, `LEXM`, `MCS-MIN`, `MCS-MAX`

CGH = BNF/GBA/fussy/default

Normal Forms

Boolean Normal Form (BNF):



Negation Normal Form (NNF):

pushing negations all the way to atomic propositions

TGBA: A New Symbolic Automaton Form

- Requires NNF
- Avoid declaring variables for *eventuality expansion rules*
  CGH/GBA: $p \mathcal{U} q \equiv$ `q | (p & VAR_X__p_U_q)`
- Ensure eventualities using *promise variables*[2]
  TGBA: $p \mathcal{U} q \equiv$ `( (q) | (p & P__p_U_q & (next(VAR__p_U_q))))`
- Simpler transitions
- Fairness == Promise fulfilled: `FAIRNESS (!P__p_U_q)`
- Correctness proof is more subtle than CGH/GBA

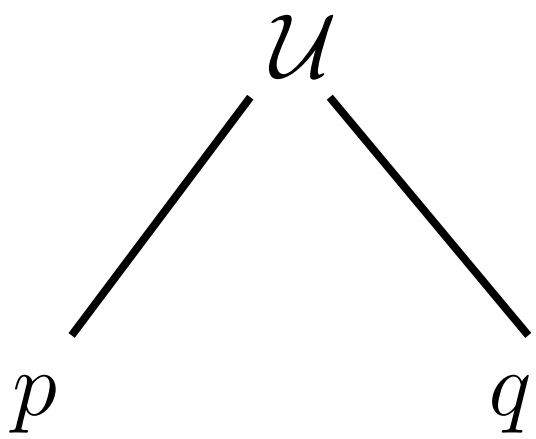
[2] based on Couvreur, On-the-Fly Verification of Linear Temporal Logic. FM'99

Sloppy: A New Transition Form

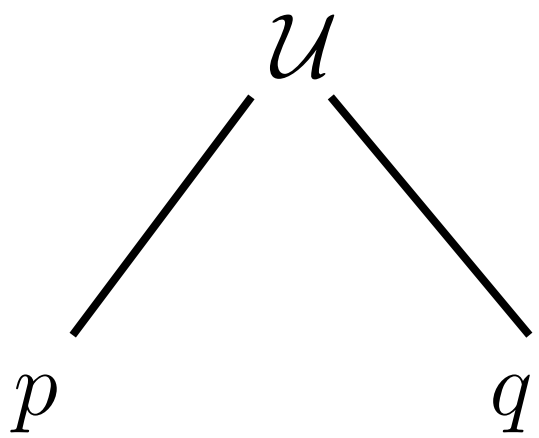
fussy	sloppy
• *single-rail encoding* • symbolic automaton has iff-transitions • `TRANS ( EL_g = (S_g) )` • BNF or NNF • more deterministic automaton	• *dual-rail encoding* • symbolic automaton has if-transitions • `TRANS ( EL_g -> (S_g) )` • requires NNF • less deterministic automaton

Variable Graph

Variable graphs formed from the parse tree for $f = (p\ \mathcal{U}\ q)$.



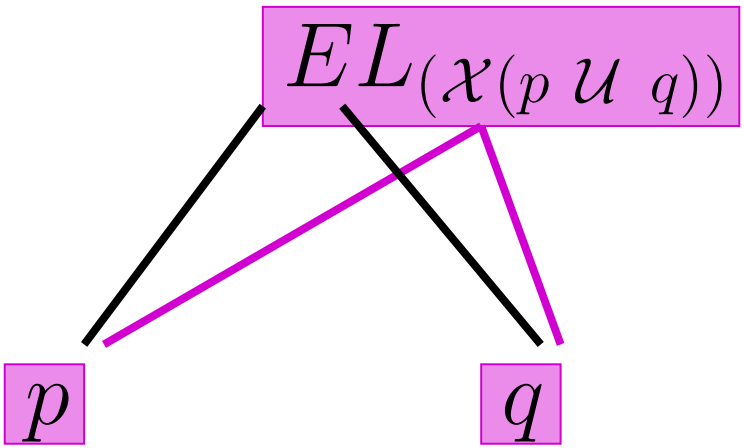
Parse Tree



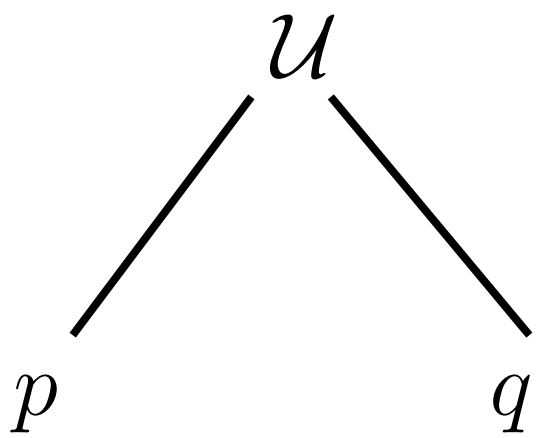
Parse Tree

Variable Graph

Variable graphs formed from the parse tree for $f = (p\ \mathcal{U}\ q)$.



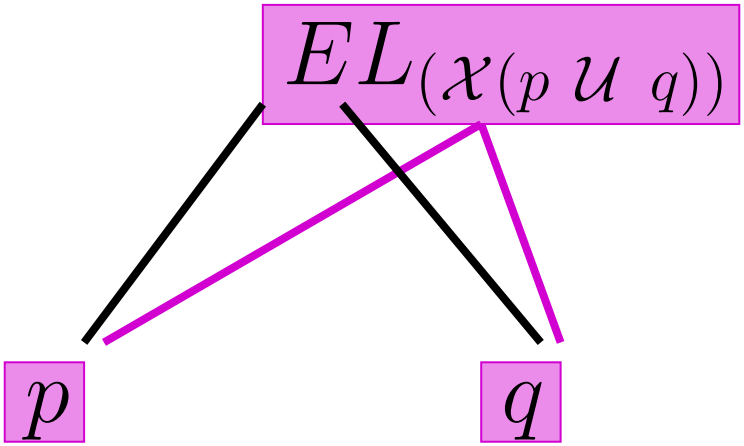
GBA Variable Graph



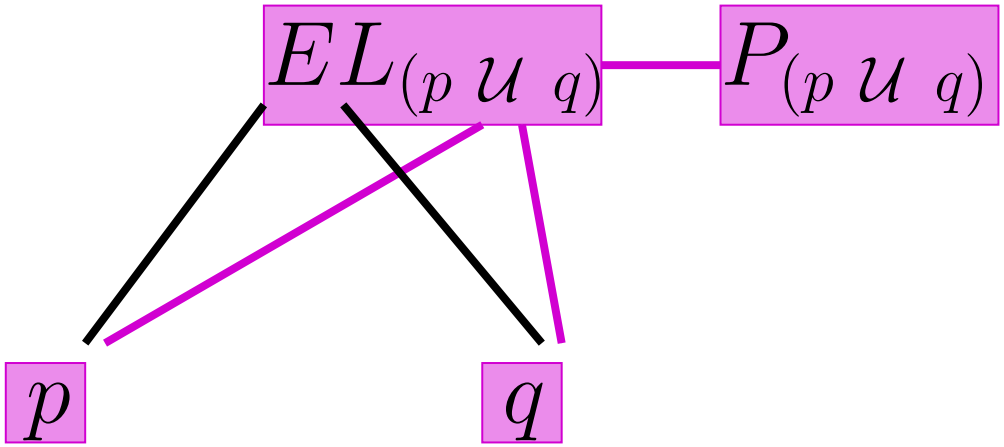
Parse Tree

Variable Graph

Variable graphs formed from the parse tree for $f = (p\ \mathcal{U}\ q)$.



GBA Variable Graph



TGBA Variable Graph

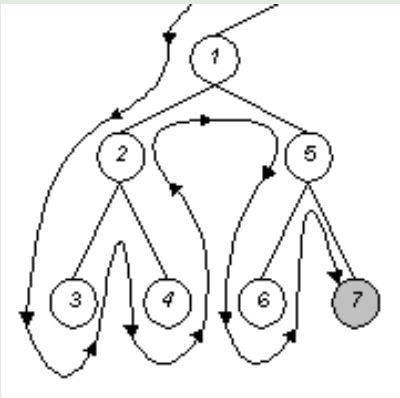
New Variable Orders

- Repurposing heuristics for bounding graph treewidth
- Ordering tree vertices based on graph triangulation algorithms

default

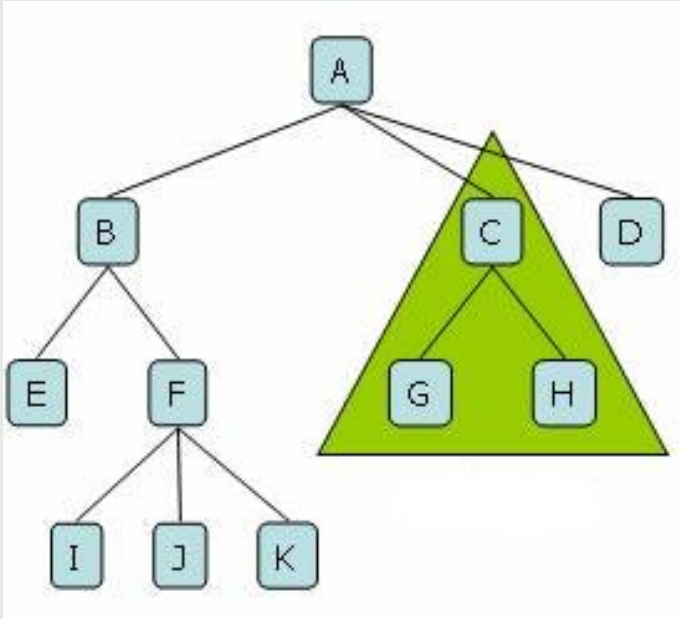
CadenceSMV default order

naïve



Pre-order, Depth First Search

New Variable Orders



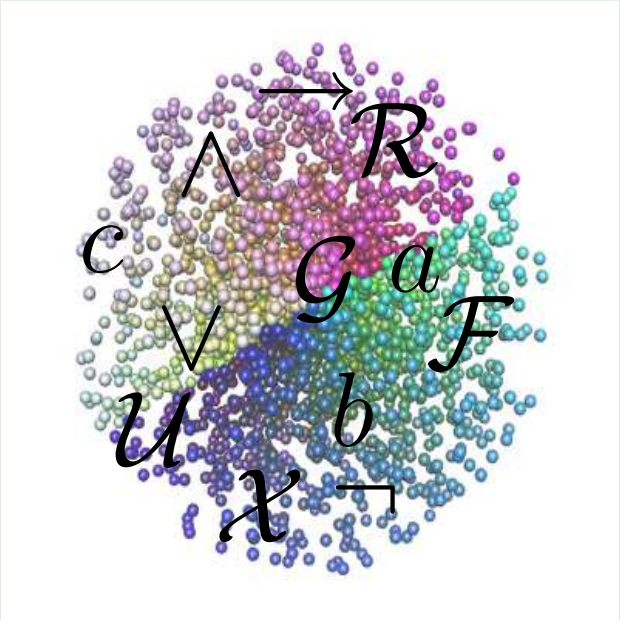
LEXP LEXM MCS-MIN MCS-MAX

30 Combinations

<table>
<tr><th>Automaton Form</th><th>Normal Form</th><th>Transition Form</th><th>Variable Order</th></tr>
<tr><td rowspan="2">GBA</td><td>BNF</td><td>fussy</td><td rowspan="6">default
naïve
LEXP
LEXM
MCS-MIN
MCS-MAX</td></tr>
<tr><td rowspan="3">NNF</td><td rowspan="2">fussy</td></tr>
<tr><td>TGBA</td></tr>
<tr><td></td><td>sloppy</td></tr>
</table>

Input Formulas[3]

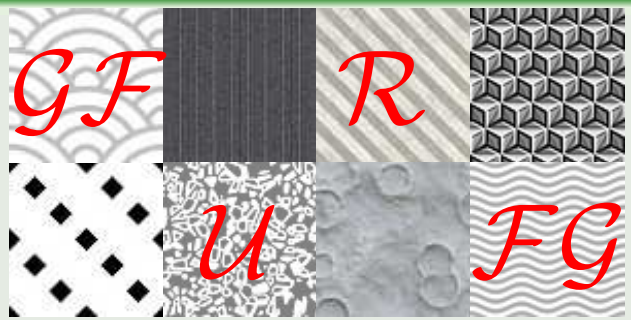
Random Formulas: 60,000



Counter Formulas: ~60 (4 types)

00 01 10 11 ...
000 001 010 011 100 ...
0000 0001 0010 0011 0100 0101 ...
00000 00001 00010 00011 00100 00101 00110 ...
⋮

Pattern Formulas: ~8,000 (9 patterns)



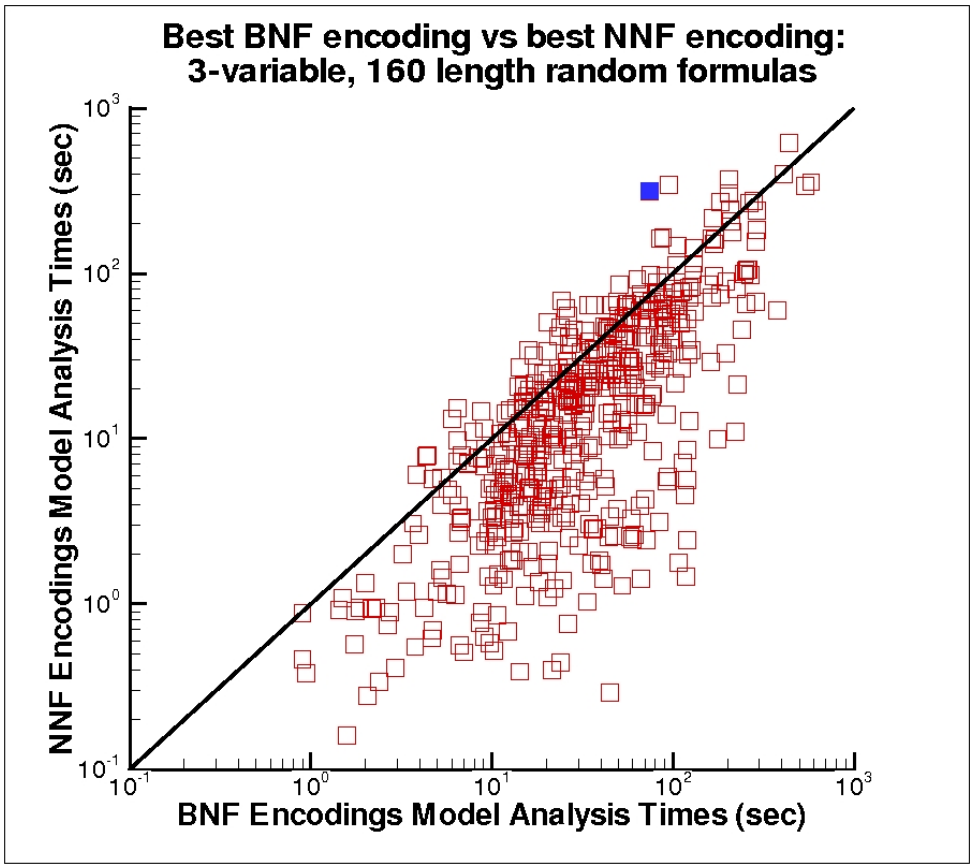
[3] Kristin Y. Rozier and Moshe Y. Vardi, LTL Satisfiability Checking. SPIN'07.

Experimental Results

1. Seven configurations are not competitive.
2. NNF is the best normal form, most (but not all) of the time.
3. No automaton form is best.
4. No transition form is best.
5. No variable order is best; `LEXM` is not competitive.
6. A formula class typically has a best encoding, but predictions are difficult.

PANDA: implements all 30 encodings

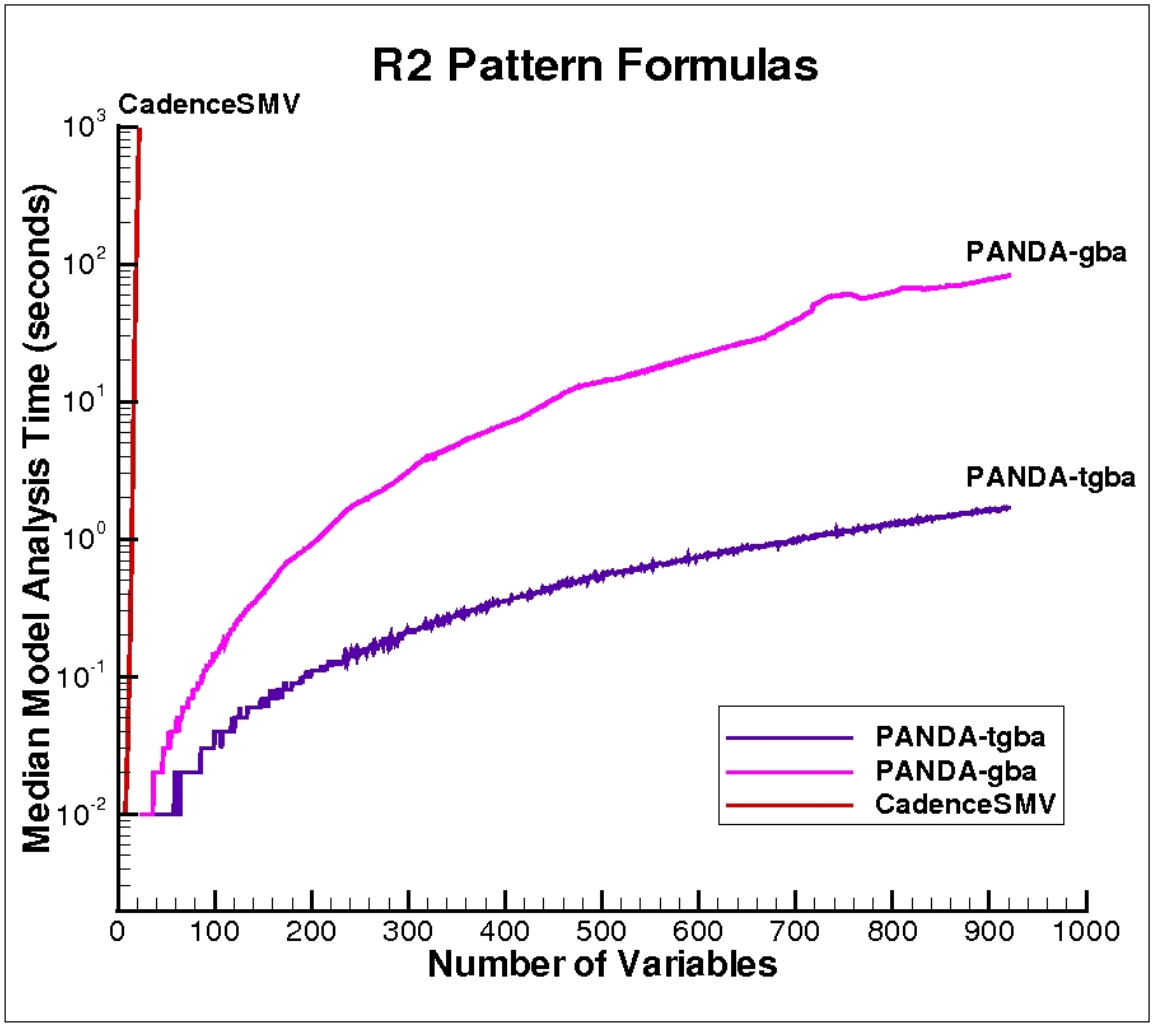
NNF is the best normal form, most (not all) of the time



Points fall below the diagonal when NNF is best.

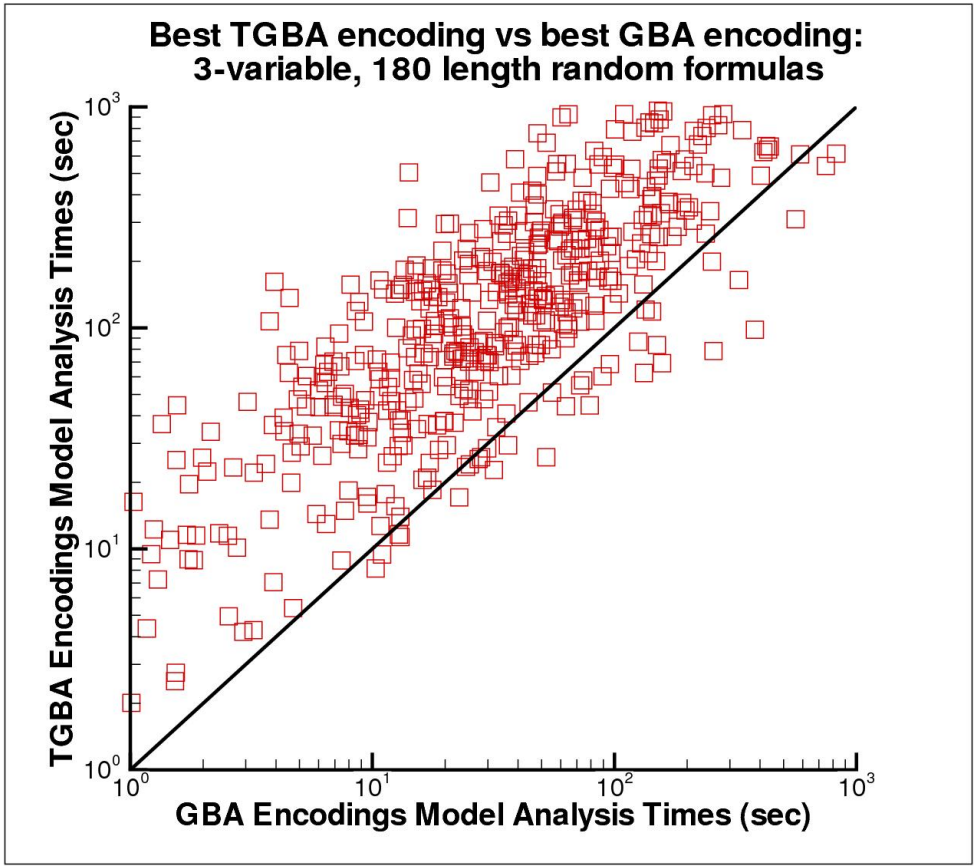
- NNF encodings were always better for all counter and pattern formulas.
- BNF encodings were optimal for a nontrivial portion of our random formulas.

TGBAs can beat CGH/CadenceSMV



$$R_2(n) = (..(p_1 \ \mathcal{R} \ p_2) \ \mathcal{R} \ \ldots) \ \mathcal{R} \ p_n.$$

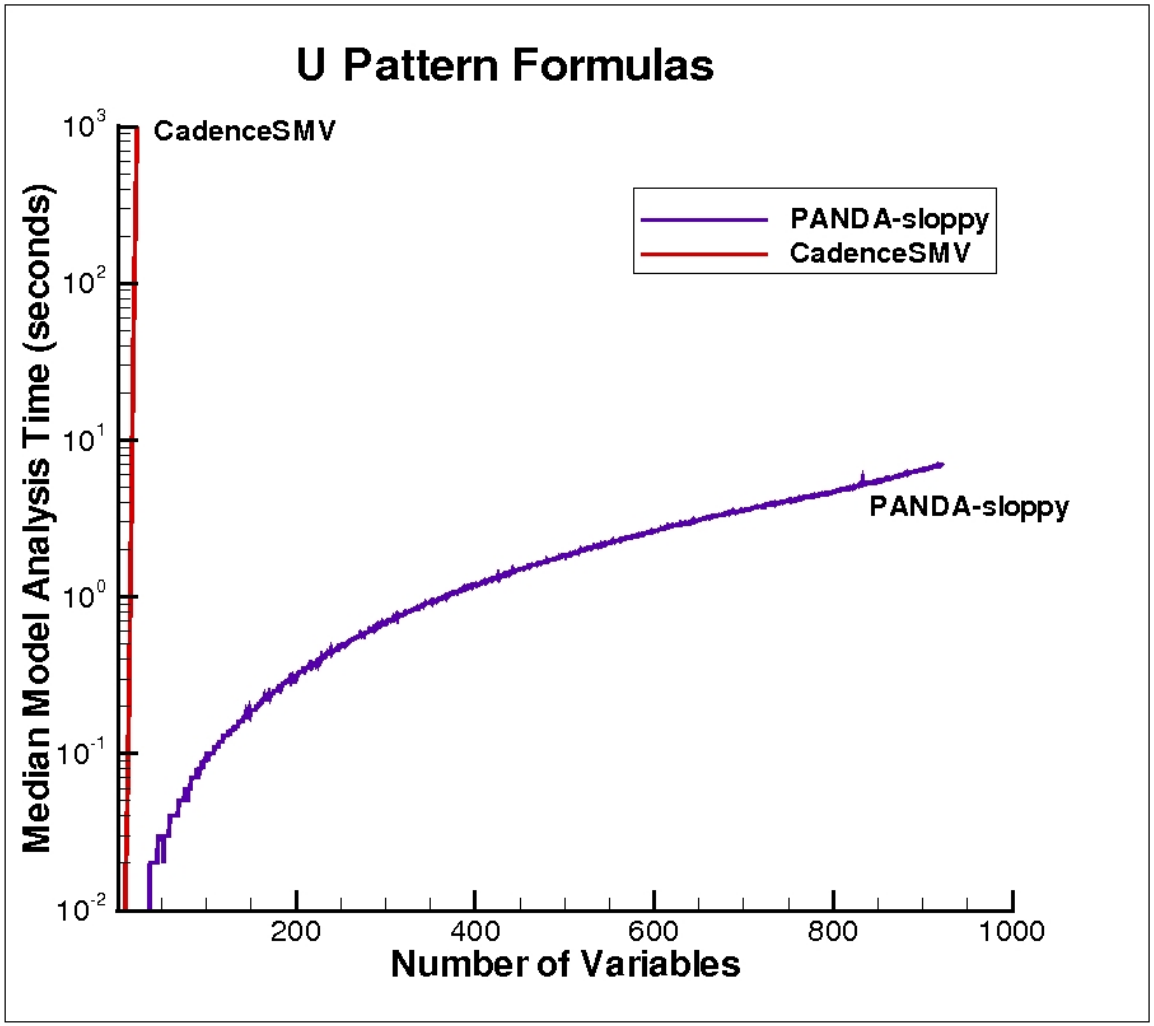
No automaton form is best



Points fall below the diagonal when TGBA is best.

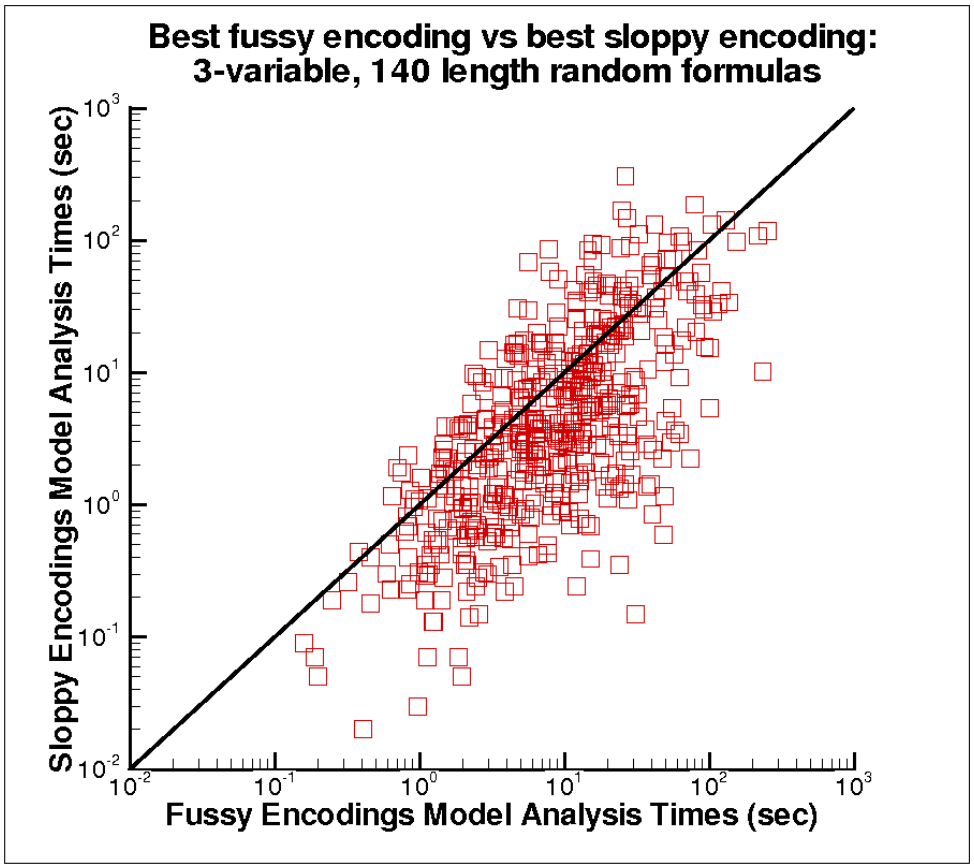
- TGBA encodings are better for $C2$, $R2$, U, and $C1$ pattern formulas.
- GBA encodings are better for R-pattern formulas, majority of random formulas.
- TGBA is better for 3-variable counters.
- GBA is better for 2-variable linear counters.

Sloppy transitions can beat CGH/CadenceSMV



$$U(n) = (\ldots(p_1 \; \mathcal{U} \; p_2) \; \mathcal{U} \; \ldots) \; \mathcal{U} \; p_n.$$

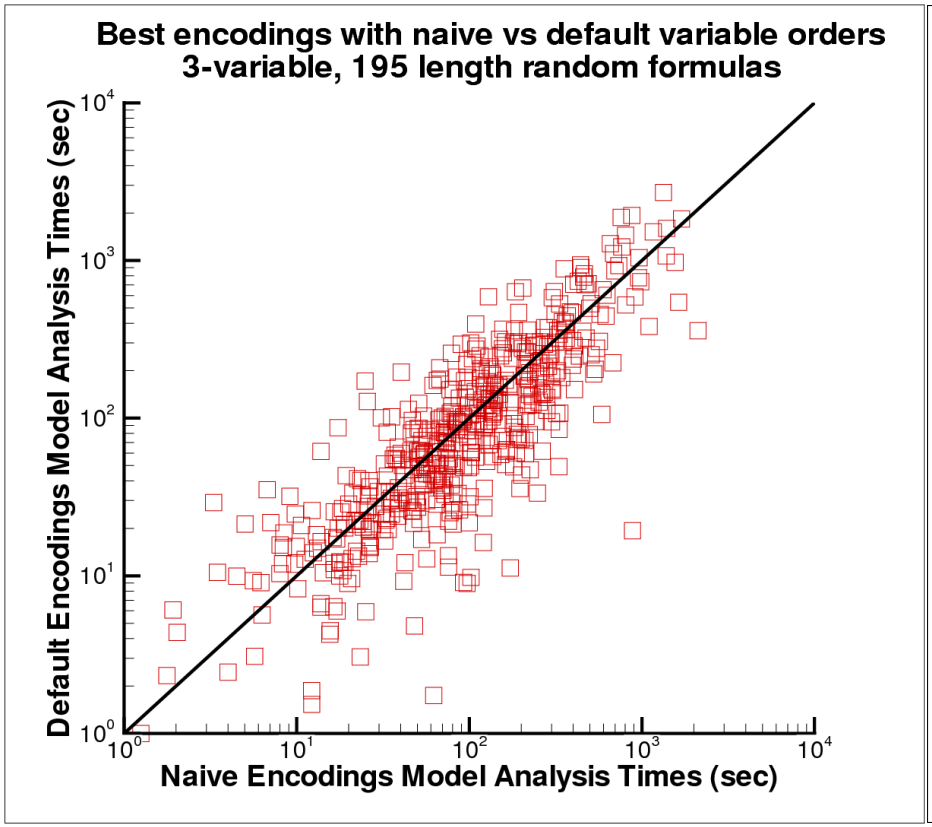
No transition form is best

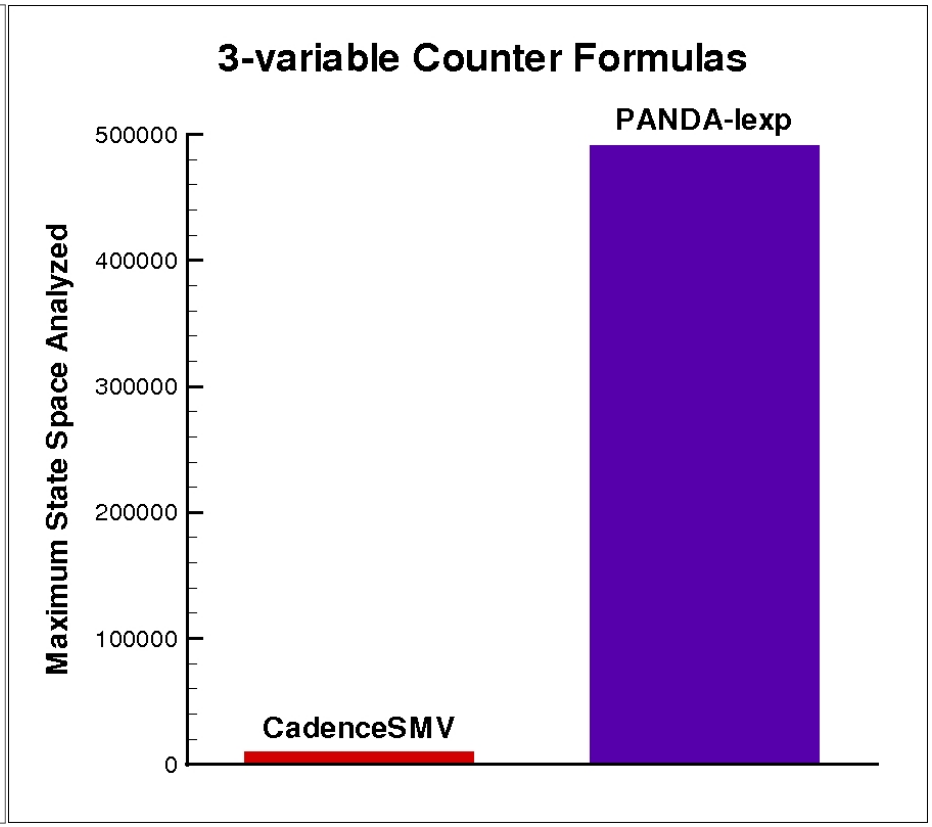


Points fall below the diagonal when sloppy encoding is best.

- Sloppy encoding is the best transition form for all pattern formulas.
- Fussy encoding is better for all counter formulas.

No variable order is best, but LEXM is worst





Solution! PANDA: A Multi-Encoding Approach

Our new tool: **PANDA** (Portfolio Approach to Navigate the Design of Automata)

- Multi-encoding approach:
  - run many PANDA encodings in parallel
  - terminate when the first job completes

Discussion

- Each of our novel encoding techniques has significant impact on performance.
- No single encoding is dominant.
- Use a multi-encoding approach: run many encodings in parallel.
- Our approach is *extensible*.

We can consistently significantly dominate the native translation of CadenceSMV.

`http://ti.arc.nasa.gov/profile/kyrozier`

Further research: model checking?